

# Hack the Virtual Memory: malloc, the heap & the program break



This is the fourth chapter in a series around virtual memory. The goal is to learn some CS basics, but in a different and more practical way.

If you missed the previous chapters, you should probably start there:

- Chapter 0: [Hack The Virtual Memory: C strings & /proc](#)
- Chapter 1: [Hack The Virtual Memory: Python bytes](#)
- Chapter 2: [Hack The Virtual Memory: Drawing the VM diagram](#)

## The heap

In this chapter we will look at the heap and `malloc` in order to answer some of the questions we ended with at the end of the [previous chapter](#):

- Why doesn't our allocated memory start at the very beginning of the heap (0x2050010 vs 02050000)? What are those first 16 bytes used for?
- Is the heap actually growing upwards?

## Prerequisites

In order to fully understand this article, you will need to know:

- The basics of the C programming language (especially pointers)
- The very basics of the Linux filesystem and the shell
- We will also use the `/proc/[pid]/maps` file (see `man proc` or read our first article [Hack The Virtual Memory, chapter 0: C strings & /proc](#))

## Environment

All scripts and programs have been tested on the following system:

- Ubuntu
  - Linux ubuntu 4.4.0-31-generic #50~14.04.1-Ubuntu SMP Wed Jul 13 01:07:32 UTC 2016  
x86\_64 x86\_64 x86\_64 GNU/Linux

Tools used:

- gcc
  - gcc (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4
- glibc 2.19 (see [version.c](#) if you need to check your glibc version)
- strace
  - strace — version 4.8

Everything we will write will be true for this system/environment, but may be different on another system

We will also go through the Linux source code. If you are on Ubuntu, you can download the sources of your current kernel by running this command:

```
apt-get source linux-image-$(uname -r)
```

malloc

malloc is the common function used to dynamically allocate memory. This memory is allocated on the “heap”.

*Note: malloc is not a system call.*

From man malloc:

```
[...] allocate dynamic memory[...]
void *malloc(size_t size);
[...]
The malloc() function allocates size bytes and returns a pointer to the allocated
memory.
```

## No malloc, no [heap]

Let’s look at memory regions of a process that does not call malloc (0-main.c).

```
#include <stdlib.h>
#include <stdio.h>

/**
 * main - do nothing
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    getchar();
    return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 0-  
main.c -o 0  
julien@holberton:~/holberton/w/hackthevm3$ ./0
```

*Quick reminder (1/3): the memory regions of a process are listed in the `/proc/[pid]/maps` file. As a result, we first need to know the PID of the process. That is done using the `ps` command; the second column of `ps aux` output will give us the PID of the process. Please read [chapter 0](#) to learn more.*

```
julien@holberton:/tmp$ ps aux | grep \ \./0$  
julien      3638  0.0  0.0  4200   648 pts/9    S+   12:01   0:00 ./0
```

*Quick reminder (2/3): from the above output, we can see that the PID of the process we want to look at is 3638. As a result, the `maps` file will be found in the directory `/proc/3638`.*

```
julien@holberton:/tmp$ cd /proc/3638
```

*Quick reminder (3/3): The `maps` file contains the memory regions of the process. The format of each line in this file is:*

*address perms offset dev inode pathname*

```
julien@holberton:/proc/3638$ cat maps  
00400000-00401000 r-xp 00000000 08:01 174583                               /home/  
julien/holberton/w/hack_the_virtual_memory/03. The Heap/0  
00600000-00601000 r--p 00000000 08:01 174583                               /home/  
julien/holberton/w/hack_the_virtual_memory/03. The Heap/0  
00601000-00602000 rw-p 00001000 08:01 174583                               /home/  
julien/holberton/w/hack_the_virtual_memory/03. The Heap/0  
7f38f87d7000-7f38f8991000 r-xp 00000000 08:01 136253                       /lib/  
x86_64-linux-gnu/libc-2.19.so  
7f38f8991000-7f38f8b91000 ---p 001ba000 08:01 136253                       /lib/  
x86_64-linux-gnu/libc-2.19.so  
7f38f8b91000-7f38f8b95000 r--p 001ba000 08:01 136253                       /lib/  
x86_64-linux-gnu/libc-2.19.so  
7f38f8b95000-7f38f8b97000 rw-p 001be000 08:01 136253                       /lib/  
x86_64-linux-gnu/libc-2.19.so  
7f38f8b97000-7f38f8b9c000 rw-p 00000000 00:00 0  
7f38f8b9c000-7f38f8bbf000 r-xp 00000000 08:01 136229                       /lib/  
x86_64-linux-gnu/ld-2.19.so  
7f38f8da3000-7f38f8da6000 rw-p 00000000 00:00 0  
7f38f8dbb000-7f38f8dbe000 rw-p 00000000 00:00 0  
7f38f8dbe000-7f38f8dbf000 r--p 00022000 08:01 136229                       /lib/  
x86_64-linux-gnu/ld-2.19.so  
7f38f8dbf000-7f38f8dc0000 rw-p 00023000 08:01 136229                       /lib/  
x86_64-linux-gnu/ld-2.19.so  
7f38f8dc0000-7f38f8dc1000 rw-p 00000000 00:00 0  
7ffdd85c5000-7ffdd85e6000 rw-p 00000000 00:00 0                          [stack]  
7ffdd85f2000-7ffdd85f4000 r--p 00000000 00:00 0                          [vvar]  
7ffdd85f4000-7ffdd85f6000 r-xp 00000000 00:00 0                          [vdso]  
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0                    [vsyscall]  
julien@holberton:/proc/3638$
```

*Note: `hackthevm3` is a symbolic link to `hack_the_virtual_memory/03`. The `Heap/`*

-> As we can see from the above `maps` file, there's no `[heap]` region allocated.

malloc(x)

Let's do the same but with a program that calls malloc (1-main.c):

```
#include <stdio.h>
#include <stdlib.h>

/**
 * main - 1 call to malloc
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    malloc(1);
    getchar();
    return (EXIT_SUCCESS);
}
```

```
julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 1-
main.c -o 1
julien@holberton:~/holberton/w/hackthevm3$ ./1
```

```
julien@holberton:/proc/3638$ ps aux | grep \ \./1$
julien      3718  0.0  0.0  4332  660 pts/9    S+   12:09   0:00 ./1
julien@holberton:/proc/3638$ cd /proc/3718
julien@holberton:/proc/3718$ cat maps
00400000-00401000 r-xp 00000000 08:01 176964                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/1
00600000-00601000 r--p 00000000 08:01 176964                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/1
00601000-00602000 rw-p 00001000 08:01 176964                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/1
01195000-011b6000 rw-p 00000000 00:00 0                    [heap]
...
julien@holberton:/proc/3718$
```

-> the [heap] is here.

Let's check the return value of malloc to make sure the returned address is in the heap region (2-main.c):

```

#include <stdio.h>
#include <stdlib.h>

/**
 * main - prints the malloc returned address
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    void *p;

    p = malloc(1);
    printf("%p\n", p);
    getchar();
    return (EXIT_SUCCESS);
}

```

```

julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 2-
main.c -o 2
julien@holberton:~/holberton/w/hackthevm3$ ./2
0x24d6010

```

```

julien@holberton:/proc/3718$ ps aux | grep \ \./2$
julien      3834  0.0  0.0  4336   676 pts/9    S+   12:48   0:00 ./2
julien@holberton:/proc/3718$ cd /proc/3834
julien@holberton:/proc/3834$ cat maps
00400000-00401000 r-xp 00000000 08:01 176966                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/2
00600000-00601000 r--p 00000000 08:01 176966                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/2
00601000-00602000 rw-p 00001000 08:01 176966                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/2
024d6000-024f7000 rw-p 00000000 00:00 0                    [heap]
...
julien@holberton:/proc/3834$

```

-> 024d6000 < 0x24d6010 < 024f7000

The returned address is inside the heap region. And as we have seen in the [previous chapter](#), the returned address does not start exactly at the beginning of the region; we'll see why later.

strace, brk **and** sbrk

malloc is a “regular” function (as opposed to a system call), so it must call some kind of syscall in order to manipulate the heap. Let's use strace to find out.

strace is a program used to trace system calls and signals. Any program will always use a few syscalls before your main function is executed. In order to know which syscalls are used by malloc, we will add a write syscall before and after the call to malloc(3-main.c).

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/**
 * main - let's find out which syscall malloc is using
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    void *p;

    write(1, "BEFORE MALLOC\n", 14);
    p = malloc(1);
    write(1, "AFTER MALLOC\n", 13);
    printf("%p\n", p);
    getchar();
    return (EXIT_SUCCESS);
}

```

```

julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 3-
main.c -o 3
julien@holberton:~/holberton/w/hackthevm3$ strace ./3
execve("./3", [". /3"], [/ * 61 vars */]) = 0
...
write(1, "BEFORE MALLOC\n", 14BEFORE MALLOC
)          = 14
brk(0)                                = 0xe70000
brk(0xe91000)                          = 0xe91000
write(1, "AFTER MALLOC\n", 13AFTER MALLOC
)          = 13
...
read(0,

```

From the above listing we can focus on this:

```

brk(0)                                = 0xe70000
brk(0xe91000)                          = 0xe91000

```

-> malloc is using the brk system call in order to manipulate the heap. From brk man page (man brk), we can see what this system call is doing:

```
...
int brk(void *addr);
void *sbrk(intptr_t increment);
...
```

#### DESCRIPTION

`brk()` and `sbrk()` change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

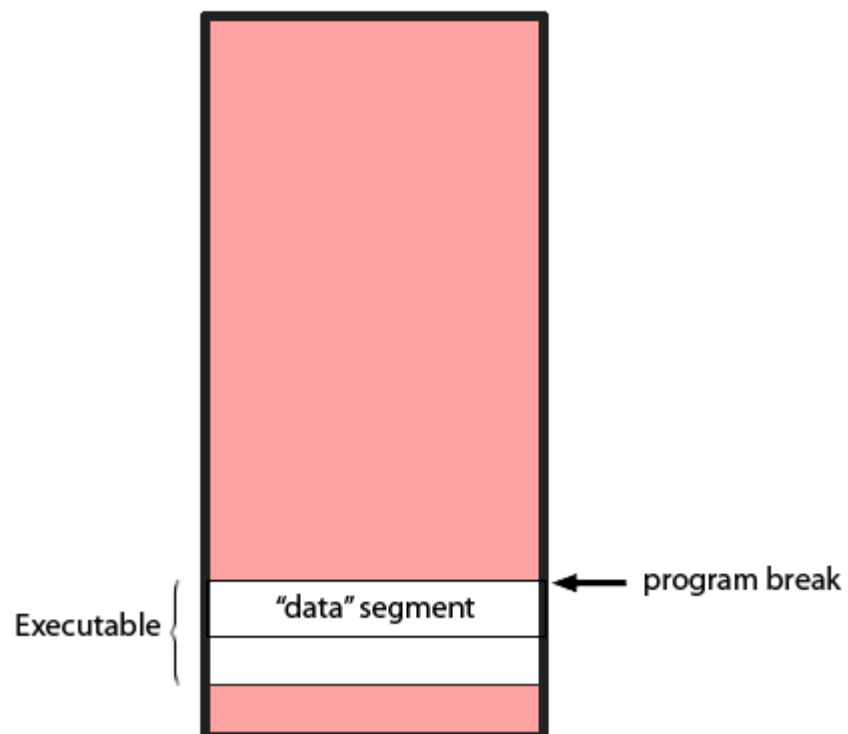
`brk()` sets the end of the data segment to the value specified by `addr`, when that value is reasonable, the system has enough memory, and the process does not exceed its maximum data size (see `setrlimit(2)`).

`sbrk()` increments the program's data space by `increment` bytes. Calling `sbrk()` with an increment of 0 can be used to find the current location of the program break.

The program break is the address of the first location beyond the current end of the data region of the program in the virtual memory.



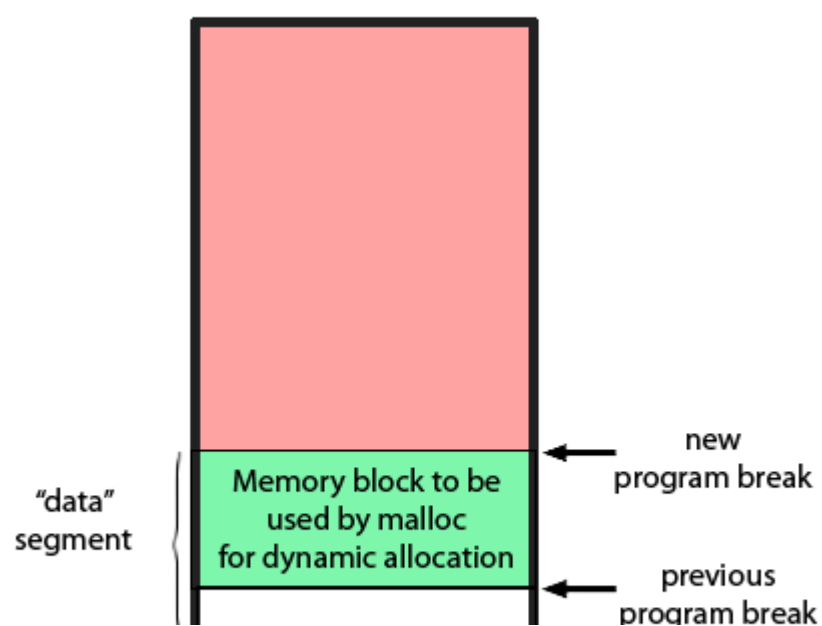
## THE VIRTUAL MEMORY



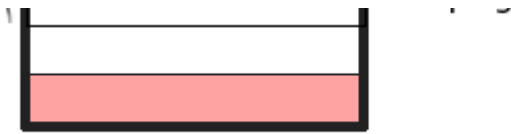
By increasing the value of the program break, via `brk` or `sbrk`, the function `malloc` creates a new space that can then be used by the process to dynamically allocate memory (using `malloc`).



## THE VIRTUAL MEMORY







So the heap is actually an extension of the data segment of the program.

The first call to `brk (brk(0))` returns the current address of the program break to `malloc`. And the second call is the one that actually creates new memory (since `0xe91000 > 0xe70000`) by increasing the value of the program break. In the above example, the heap is now starting at `0xe70000` and ends at `0xe91000`. Let's double check with the `/proc/[PID]/maps` file:

```
julien@holberton:/proc/3855$ ps aux | grep \ ./3$
julien      4011  0.0  0.0  4748   708 pts/9    S+   13:04   0:00 strace ./3
julien      4014  0.0  0.0  4336   644 pts/9    S+   13:04   0:00 ./3
julien@holberton:/proc/3855$ cd /proc/4014
julien@holberton:/proc/4014$ cat maps
00400000-00401000 r-xp 00000000 08:01 176967                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/3
00600000-00601000 r--p 00000000 08:01 176967                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/3
00601000-00602000 rw-p 00001000 08:01 176967                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/3
00e70000-00e91000 rw-p 00000000 00:00 0                    [heap]
...
julien@holberton:/proc/4014$
```

-> `00e70000-00e91000 rw-p 00000000 00:00 0 [heap]` matches the pointers returned back to `malloc` by `brk`.

That's great, but wait, why did `malloc` increment the heap by `00e91000 - 00e70000 = 0x21000` or 135168 bytes, when we only asked for only 1 byte?

## Many mallocs

What will happen if we call `malloc` several times? (4-main.c)



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/**
 * main - many calls to malloc
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    void *p;

    write(1, "BEFORE MALLOC #0\n", 17);
    p = malloc(1024);
    write(1, "AFTER MALLOC #0\n", 16);
    printf("%p\n", p);

    write(1, "BEFORE MALLOC #1\n", 17);
    p = malloc(1024);
    write(1, "AFTER MALLOC #1\n", 16);
    printf("%p\n", p);

    write(1, "BEFORE MALLOC #2\n", 17);
    p = malloc(1024);
    write(1, "AFTER MALLOC #2\n", 16);
    printf("%p\n", p);

    write(1, "BEFORE MALLOC #3\n", 17);
    p = malloc(1024);
    write(1, "AFTER MALLOC #3\n", 16);
    printf("%p\n", p);

    getchar();
    return (EXIT_SUCCESS);
}
```

```

julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 4-
main.c -o 4
julien@holberton:~/holberton/w/hackthevm3$ strace ./4
execve("./4", ["/4"], [/* 61 vars */]) = 0
...
write(1, "BEFORE MALLOC #0\n", 17BEFORE MALLOC #0
)      = 17
brk(0)                                = 0x1314000
brk(0x1335000)                        = 0x1335000
write(1, "AFTER MALLOC #0\n", 16AFTER MALLOC #0
)      = 16
...
write(1, "0x1314010\n", 100x1314010
)      = 10
write(1, "BEFORE MALLOC #1\n", 17BEFORE MALLOC #1
)      = 17
write(1, "AFTER MALLOC #1\n", 16AFTER MALLOC #1
)      = 16
write(1, "0x1314420\n", 100x1314420
)      = 10
write(1, "BEFORE MALLOC #2\n", 17BEFORE MALLOC #2
)      = 17
write(1, "AFTER MALLOC #2\n", 16AFTER MALLOC #2
)      = 16
write(1, "0x1314830\n", 100x1314830
)      = 10
write(1, "BEFORE MALLOC #3\n", 17BEFORE MALLOC #3
)      = 17
write(1, "AFTER MALLOC #3\n", 16AFTER MALLOC #3
)      = 16
write(1, "0x1314c40\n", 100x1314c40
)      = 10
...
read(0,

```

-> malloc is NOT calling brk each time we call it.

The first time, malloc creates a new space (the heap) for the program (by increasing the program break location). The following times, malloc uses the same space to give our program “new” chunks of memory. Those “new” chunks of memory are part of the memory previously allocated using brk. This way, malloc doesn’t have to use syscalls (brk) every time we call it, and thus it makes malloc – and our programs using malloc – faster. It also allows malloc and free to optimize the usage of the memory.

Let’s double check that we have only one heap, allocated by the first call to brk:

```

julien@holberton:/proc/4014$ ps aux | grep \ ./4$
julien      4169  0.0  0.0  4748   688 pts/9    S+   13:33   0:00 strace ./4
julien      4172  0.0  0.0  4336   656 pts/9    S+   13:33   0:00 ./4
julien@holberton:/proc/4014$ cd /proc/4172
julien@holberton:/proc/4172$ cat maps
00400000-00401000 r-xp 00000000 08:01 176973                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/4
00600000-00601000 r--p 00000000 08:01 176973                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/4
00601000-00602000 rw-p 00001000 08:01 176973                /home/
julien/holberton/w/hack_the_virtual_memory/03. The Heap/4
01314000-01335000 rw-p 00000000 00:00 0                    [heap]
7f4a3f2c4000-7f4a3f47e000 r-xp 00000000 08:01 136253          /lib/
x86_64-linux-gnu/libc-2.19.so
7f4a3f47e000-7f4a3f67e000 ---p 001ba000 08:01 136253          /lib/
x86_64-linux-gnu/libc-2.19.so
7f4a3f67e000-7f4a3f682000 r--p 001ba000 08:01 136253          /lib/
x86_64-linux-gnu/libc-2.19.so
7f4a3f682000-7f4a3f684000 rw-p 001be000 08:01 136253          /lib/
x86_64-linux-gnu/libc-2.19.so
7f4a3f684000-7f4a3f689000 rw-p 00000000 00:00 0
7f4a3f689000-7f4a3f6ac000 r-xp 00000000 08:01 136229          /lib/
x86_64-linux-gnu/ld-2.19.so
7f4a3f890000-7f4a3f893000 rw-p 00000000 00:00 0
7f4a3f8a7000-7f4a3f8ab000 rw-p 00000000 00:00 0
7f4a3f8ab000-7f4a3f8ac000 r--p 00022000 08:01 136229          /lib/
x86_64-linux-gnu/ld-2.19.so
7f4a3f8ac000-7f4a3f8ad000 rw-p 00023000 08:01 136229          /lib/
x86_64-linux-gnu/ld-2.19.so
7f4a3f8ad000-7f4a3f8ae000 rw-p 00000000 00:00 0
7ffd1ba73000-7ffd1ba94000 rw-p 00000000 00:00 0                [stack]
7ffd1bbef000-7ffd1bbef000 r--p 00000000 00:00 0                [vvar]
7ffd1bbef000-7ffd1bbf1000 r-xp 00000000 00:00 0                [vdso]
fffffffff600000-fffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
julien@holberton:/proc/4172$

```

-> We have only one [heap] and the addresses match those returned by sbrk: 0x1314000 & 0x1335000

## Naive malloc

Based on the above, and assuming we won't ever need to free anything, we can now write our own (naive) version of `malloc`, that would move the program break each time it is called.

```

#include <stdlib.h>
#include <unistd.h>

/**
 * malloc - naive version of malloc: dynamically allocates memory on the heap using
 sbrk
 * @size: number of bytes to allocate
 *
 * Return: the memory address newly allocated, or NULL on error
 *
 * Note: don't do this at home :)
 */
void *malloc(size_t size)
{
    void *previous_break;

    previous_break = sbrk(size);
    /* check for error */
    if (previous_break == (void *) -1)
    {
        /* on error malloc returns NULL */
        return (NULL);
    }
    return (previous_break);
}

```

## The 0x10 lost bytes

If we look at the output of the previous program (4-main.c), we can see that the first memory address returned by `malloc` doesn't start at the beginning of the heap, but 0x10 bytes after: 0x1314010 vs 0x1314000. Also, when we call `malloc(1024)` a second time, the address should be 0x1314010 (the returned value of the first call to `malloc`) + 1024 (or 0x400 in hexadecimal, since the first call to `malloc` was asking for 1024 bytes) = 0x1318010. But the return value of the second call to `malloc` is 0x1314420. We have lost 0x10 bytes again! Same goes for the subsequent calls.

Let's look at what we can find inside those "lost" 0x10-byte memory spaces (5-main.c) and whether the memory loss stays constant:

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/**
 * pmem - print mem
 * @p: memory address to start printing from
 * @bytes: number of bytes to print
 *
 * Return: nothing
 */
void pmem(void *p, unsigned int bytes)
{
    unsigned char *ptr;
    unsigned int i;

    ptr = (unsigned char *)p;
    for (i = 0; i < bytes; i++)
    {
        if (i != 0)
        {
            printf(" ");
        }
        printf("%02x", *(ptr + i));
    }
    printf("\n");
}

/**
 * main - the 0x10 lost bytes
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    void *p;
    int i;

    for (i = 0; i < 10; i++)
    {
        p = malloc(1024 * (i + 1));
        printf("%p\n", p);
        printf("bytes at %p:\n", (void *)((char *)p - 0x10));
        pmem((char *)p - 0x10, 0x10);
    }
    return (EXIT_SUCCESS);
}

```

```

julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 5-
main.c -o 5
julien@holberton:~/holberton/w/hackthevm3$ ./5
0x1fa8010
bytes at 0x1fa8000:
00 00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
0x1fa8420
bytes at 0x1fa8410:
00 00 00 00 00 00 00 00 00 11 08 00 00 00 00 00 00
0x1fa8c30
bytes at 0x1fa8c20:
00 00 00 00 00 00 00 00 00 11 0c 00 00 00 00 00 00
0x1fa9840
bytes at 0x1fa9830:
00 00 00 00 00 00 00 00 00 11 10 00 00 00 00 00 00
0x1faa850
bytes at 0x1faa840:
00 00 00 00 00 00 00 00 00 11 14 00 00 00 00 00 00
0x1fab60
bytes at 0x1fab50:
00 00 00 00 00 00 00 00 00 11 18 00 00 00 00 00 00
0x1fad470
bytes at 0x1fad460:
00 00 00 00 00 00 00 00 00 11 1c 00 00 00 00 00 00
0x1faf080
bytes at 0x1faf070:
00 00 00 00 00 00 00 00 00 11 20 00 00 00 00 00 00
0x1fb1090
bytes at 0x1fb1080:
00 00 00 00 00 00 00 00 00 11 24 00 00 00 00 00 00
0x1fb34a0
bytes at 0x1fb3490:
00 00 00 00 00 00 00 00 00 11 28 00 00 00 00 00 00
julien@holberton:~/holberton/w/hackthevm3$

```

There is one clear pattern: the size of the malloc'ed memory chunk is always found in the preceding 0x10 bytes. For instance, the first malloc call is malloc'ing 1024 (0x0400) bytes and we can find 11 04 00 00 00 00 00 00 in the preceding 0x10 bytes. Those last bytes represent the number 0x 00 00 00 00 00 00 04 11 = 0x400 (1024) + 0x10 (the block size preceding those 1024 bytes + 1 (we'll talk about this "+1" later in this chapter). If we look at each 0x10 bytes preceding the addresses returned by malloc, they all contain the size of the chunk of memory asked to malloc + 0x10 + 1.

At this point, given what we said and saw earlier, we can probably guess that those 0x10 bytes are a sort of data structure used by malloc (and free) to deal with the heap. And indeed, even though we don't understand everything yet, we can already use this data structure to go from one malloc'ed chunk of memory to the other (6-main.c) as long as we have the address of the beginning of the heap (*and as long as we have never called free*):

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/**
 * pmem - print mem
 * @p: memory address to start printing from
 * @bytes: number of bytes to print
 *
 * Return: nothing
 */
void pmem(void *p, unsigned int bytes)
{
    unsigned char *ptr;
    unsigned int i;

    ptr = (unsigned char *)p;
    for (i = 0; i < bytes; i++)
    {
        if (i != 0)
        {
            printf(" ");
        }
        printf("%02x", *(ptr + i));
    }
    printf("\n");
}

/**
 * main - using the 0x10 bytes to jump to next malloc'ed chunks
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    void *p;
    int i;
    void *heap_start;
    size_t size_of_the_block;

    heap_start = sbrk(0);
    write(1, "START\n", 6);
    for (i = 0; i < 10; i++)
    {
        p = malloc(1024 * (i + 1));
        *((int *)p) = i;
        printf("%p: [%i]\n", p, i);
    }
    p = heap_start;
    for (i = 0; i < 10; i++)
    {
        pmem(p, 0x10);
        size_of_the_block = *((size_t *)((char *)p + 8)) - 1;
        printf("%p: [%i] - size = %lu\n",
            (void *)((char *)p + 0x10),
            *((int *)((char *)p + 0x10)),
            size_of_the_block);
    }
}

```



```

        p = (void *)((char *)p + size_of_the_block);
    }
    write(1, "END\n", 4);
    return (EXIT_SUCCESS);
}

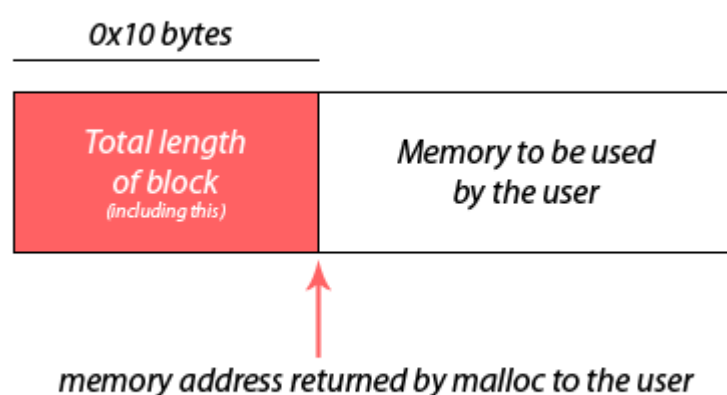
```

```

julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 6-
main.c -o 6
julien@holberton:~/holberton/w/hackthevm3$ ./6
START
0x9e6010: [0]
0x9e6420: [1]
0x9e6c30: [2]
0x9e7840: [3]
0x9e8850: [4]
0x9e9c60: [5]
0x9eb470: [6]
0x9ed080: [7]
0x9ef090: [8]
0x9f14a0: [9]
00 00 00 00 00 00 00 00 11 04 00 00 00 00 00 00
0x9e6010: [0] - size = 1040
00 00 00 00 00 00 00 00 11 08 00 00 00 00 00 00
0x9e6420: [1] - size = 2064
00 00 00 00 00 00 00 00 11 0c 00 00 00 00 00 00
0x9e6c30: [2] - size = 3088
00 00 00 00 00 00 00 00 11 10 00 00 00 00 00 00
0x9e7840: [3] - size = 4112
00 00 00 00 00 00 00 00 11 14 00 00 00 00 00 00
0x9e8850: [4] - size = 5136
00 00 00 00 00 00 00 00 11 18 00 00 00 00 00 00
0x9e9c60: [5] - size = 6160
00 00 00 00 00 00 00 00 11 1c 00 00 00 00 00 00
0x9eb470: [6] - size = 7184
00 00 00 00 00 00 00 00 11 20 00 00 00 00 00 00
0x9ed080: [7] - size = 8208
00 00 00 00 00 00 00 00 11 24 00 00 00 00 00 00
0x9ef090: [8] - size = 9232
00 00 00 00 00 00 00 00 11 28 00 00 00 00 00 00
0x9f14a0: [9] - size = 10256
END
julien@holberton:~/holberton/w/hackthevm3$

```

One of our open questions from the previous chapter is now answered: `malloc` is using 0x10 additional bytes for each `malloc`'ed memory block to store the size of the block.



This data will actually be used by `free` to save it to a list of available blocks for future calls to `malloc`.

But our study also raises a new question: what are the first 8 bytes of the 16 (`0x10` in hexadecimal) bytes used for? It seems to always be zero. Is it just padding?

## RTFSC

At this stage, we probably want to check the source code of `malloc` to confirm what we just found (`malloc.c` from the `glibc`).

```

1055 /*
1056     malloc_chunk details:
1057
1058     (The following includes lightly edited explanations by Colin Plumb.)
1059
1060     Chunks of memory are maintained using a `boundary tag' method as
1061     described in e.g., Knuth or Standish. (See the paper by Paul
1062     Wilson ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps for a
1063     survey of such techniques.) Sizes of free chunks are stored both
1064     in the front of each chunk and at the end. This makes
1065     consolidating fragmented chunks into bigger chunks very fast. The
1066     size fields also hold bits representing whether chunks are free or
1067     in use.
1068
1069     An allocated chunk looks like this:
1070
1071     chunk-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+-+
1072     |                                     Size of previous chunk, if unallocated (P clear)
1073     |
1074     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+-+
1075     |                                     Size of chunk, in bytes                                     |A|M|
P|
1076     mem-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+-+
1077     |                                     User data starts here...
1078     .
1079     .
1080     .                                     (malloc_usable_size() bytes)
1081     .
1082     |
1083     nextchunk-> +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+-+
1084     |                                     (size of chunk, but used for application data)
1085     |
1086     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+-+
1087     |                                     Size of next chunk, in bytes                                     |A|0|
1|
1088     +---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
+-+
1089
1090     Where "chunk" is the front of the chunk for the purpose of most of
1091     the malloc code, but "mem" is the pointer that is returned to the
1092     user. "Nextchunk" is the beginning of the next contiguous chunk.

```

-> We were correct \o/. Right before the address returned by malloc to the user, we have two variables:

- Size of previous chunk, if unallocated: we never free'd any chunks so that is why it was always 0
- Size of chunk, in bytes

Let's free some chunks to confirm that the first 8 bytes are used the way the source code describes it (7-main.c):

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/**
 * pmem - print mem
 * @p: memory address to start printing from
 * @bytes: number of bytes to print
 *
 * Return: nothing
 */
void pmem(void *p, unsigned int bytes)
{
    unsigned char *ptr;
    unsigned int i;

    ptr = (unsigned char *)p;
    for (i = 0; i < bytes; i++)
    {
        if (i != 0)
        {
            printf(" ");
        }
        printf("%02x", *(ptr + i));
    }
    printf("\n");
}

/**
 * main - confirm the source code
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    void *p;
    int i;
    size_t size_of_the_chunk;
    size_t size_of_the_previous_chunk;
    void *chunks[10];

    for (i = 0; i < 10; i++)
    {
        p = malloc(1024 * (i + 1));
        chunks[i] = (void *)((char *)p - 0x10);
        printf("%p\n", p);
    }
    free((char *)(chunks[3]) + 0x10);
    free((char *)(chunks[7]) + 0x10);
    for (i = 0; i < 10; i++)
    {
        p = chunks[i];
        printf("chunks[%d]: ", i);
        pmem(p, 0x10);
        size_of_the_chunk = *((size_t *)((char *)p + 8)) - 1;
        size_of_the_previous_chunk = *((size_t *)((char *)p));
        printf("chunks[%d]: %p, size = %li, prev = %li\n",

```

```

        i, p, size_of_the_chunk, size_of_the_previous_chunk);
    }
    return (EXIT_SUCCESS);
}

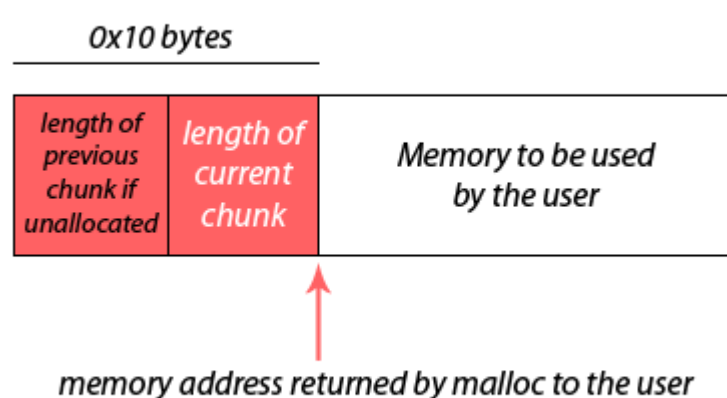
```

```

julien@holberton:~/holberton/w/hackthetvm3$ gcc -Wall -Wextra -pedantic -Werror 7-
main.c -o 7
julien@holberton:~/holberton/w/hackthetvm3$ ./7
0x1536010
0x1536420
0x1536c30
0x1537840
0x1538850
0x1539c60
0x153b470
0x153d080
0x153f090
0x15414a0
chunks[0]: 00 00 00 00 00 00 00 00 11 04 00 00 00 00 00
chunks[0]: 0x1536000, size = 1040, prev = 0
chunks[1]: 00 00 00 00 00 00 00 00 11 08 00 00 00 00 00
chunks[1]: 0x1536410, size = 2064, prev = 0
chunks[2]: 00 00 00 00 00 00 00 00 11 0c 00 00 00 00 00
chunks[2]: 0x1536c20, size = 3088, prev = 0
chunks[3]: 00 00 00 00 00 00 00 00 11 10 00 00 00 00 00
chunks[3]: 0x1537830, size = 4112, prev = 0
chunks[4]: 10 10 00 00 00 00 00 00 10 14 00 00 00 00 00
chunks[4]: 0x1538840, size = 5135, prev = 4112
chunks[5]: 00 00 00 00 00 00 00 00 11 18 00 00 00 00 00
chunks[5]: 0x1539c50, size = 6160, prev = 0
chunks[6]: 00 00 00 00 00 00 00 00 11 1c 00 00 00 00 00
chunks[6]: 0x153b460, size = 7184, prev = 0
chunks[7]: 00 00 00 00 00 00 00 00 11 20 00 00 00 00 00
chunks[7]: 0x153d070, size = 8208, prev = 0
chunks[8]: 10 20 00 00 00 00 00 00 10 24 00 00 00 00 00
chunks[8]: 0x153f080, size = 9231, prev = 8208
chunks[9]: 00 00 00 00 00 00 00 00 11 28 00 00 00 00 00
chunks[9]: 0x1541490, size = 10256, prev = 0
julien@holberton:~/holberton/w/hackthetvm3$

```

As we can see from the above listing, when the previous chunk has been free'd, the malloc chunk's first 8 bytes contain the size of the previous unallocated chunk. So the correct representation of a malloc chunk is the following:



Also, it seems that the first bit of the next 8 bytes (containing the size of the current chunk) serves as a

flag to check if the previous chunk is used (1) or not (0). So the correct updated version of our program should be written this way (8-main.c):

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/**
 * pmem - print mem
 * @p: memory address to start printing from
 * @bytes: number of bytes to print
 *
 * Return: nothing
 */
void pmem(void *p, unsigned int bytes)
{
    unsigned char *ptr;
    unsigned int i;

    ptr = (unsigned char *)p;
    for (i = 0; i < bytes; i++)
    {
        if (i != 0)
        {
            printf(" ");
        }
        printf("%02x", *(ptr + i));
    }
    printf("\n");
}

/**
 * main - updating with correct checks
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    void *p;
    int i;
    size_t size_of_the_chunk;
    size_t size_of_the_previous_chunk;
    void *chunks[10];
    char prev_used;

    for (i = 0; i < 10; i++)
    {
        p = malloc(1024 * (i + 1));
        chunks[i] = (void *)((char *)p - 0x10);
    }
    free((char *)(chunks[3]) + 0x10);
    free((char *)(chunks[7]) + 0x10);
    for (i = 0; i < 10; i++)
    {
        p = chunks[i];
        printf("chunks[%d]: ", i);
        pmem(p, 0x10);
        size_of_the_chunk = *((size_t *)((char *)p + 8));
        prev_used = size_of_the_chunk & 1;
        size_of_the_chunk -= prev_used;
    }
}

```



```

        size_of_the_previous_chunk = *((size_t *)((char *)p));
        printf("chunks[%d]: %p, size = %li, prev (%s) = %li\n",
               i, p, size_of_the_chunk,
               (prev_used? "allocated": "unallocated"), size_of_the_previous_chunk);
    }
    return (EXIT_SUCCESS);
}

```

```

julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 8-
main.c -o 8
julien@holberton:~/holberton/w/hackthevm3$ ./8
chunks[0]: 00 00 00 00 00 00 00 00 00 11 04 00 00 00 00 00
chunks[0]: 0x1031000, size = 1040, prev (allocated) = 0
chunks[1]: 00 00 00 00 00 00 00 00 00 11 08 00 00 00 00 00
chunks[1]: 0x1031410, size = 2064, prev (allocated) = 0
chunks[2]: 00 00 00 00 00 00 00 00 00 11 0c 00 00 00 00 00
chunks[2]: 0x1031c20, size = 3088, prev (allocated) = 0
chunks[3]: 00 00 00 00 00 00 00 00 00 11 10 00 00 00 00 00
chunks[3]: 0x1032830, size = 4112, prev (allocated) = 0
chunks[4]: 10 10 00 00 00 00 00 00 00 10 14 00 00 00 00 00
chunks[4]: 0x1033840, size = 5136, prev (unallocated) = 4112
chunks[5]: 00 00 00 00 00 00 00 00 00 11 18 00 00 00 00 00
chunks[5]: 0x1034c50, size = 6160, prev (allocated) = 0
chunks[6]: 00 00 00 00 00 00 00 00 00 11 1c 00 00 00 00 00
chunks[6]: 0x1036460, size = 7184, prev (allocated) = 0
chunks[7]: 00 00 00 00 00 00 00 00 00 11 20 00 00 00 00 00
chunks[7]: 0x1038070, size = 8208, prev (allocated) = 0
chunks[8]: 10 20 00 00 00 00 00 00 00 10 24 00 00 00 00 00
chunks[8]: 0x103a080, size = 9232, prev (unallocated) = 8208
chunks[9]: 00 00 00 00 00 00 00 00 00 11 28 00 00 00 00 00
chunks[9]: 0x103c490, size = 10256, prev (allocated) = 0
julien@holberton:~/holberton/w/hackthevm3$

```

## Is the heap actually growing upwards?

The last question left unanswered is: “Is the heap actually growing upwards?”. From the `brk` man page, it seems so:

### DESCRIPTION

`brk()` and `sbrk()` change the location of the program break, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

Let's check! (9-main.c)

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/**
 * main - moving the program break
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    int i;

    write(1, "START\n", 6);
    malloc(1);
    getchar();
    write(1, "LOOP\n", 5);
    for (i = 0; i < 0x25000 / 1024; i++)
    {
        malloc(1024);
    }
    write(1, "END\n", 4);
    getchar();
    return (EXIT_SUCCESS);
}

```

Now let's confirm this assumption with strace:

```

julien@holberton:~/holberton/w/hackthevm3$ strace ./9
execve("./9", ["../9"], [/* 61 vars */]) = 0
...
write(1, "START\n", 6START
)
= 6
brk(0)
= 0x1fd8000
brk(0x1ff9000)
= 0x1ff9000
...
write(1, "LOOP\n", 5LOOP
)
= 5
brk(0x201a000)
= 0x201a000
write(1, "END\n", 4END
)
= 4
...
julien@holberton:~/holberton/w/hackthevm3$

```

clearly, malloc made only two calls to brk to increase the allocated space on the heap. And the second call is using a higher memory address argument (0x201a000 > 0x1ff9000). The second syscall was triggered when the space on the heap was too small to host all the malloc calls.

Let's double check with /proc.

```

julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 9-
main.c -o 9
julien@holberton:~/holberton/w/hackthevm3$ ./9
START

```

```
julien@holberton:/proc/7855$ ps aux | grep \ \./9$
julien      7972  0.0  0.0  4332   684 pts/9    S+   19:08   0:00 ./9
julien@holberton:/proc/7855$ cd /proc/7972
julien@holberton:/proc/7972$ cat maps

...
00901000-00922000 rw-p 00000000 00:00 0                                [heap]
...
julien@holberton:/proc/7972$
```

-> 00901000-00922000 rw-p 00000000 00:00 0 [heap]

Let's hit Enter and look at the [heap] again:

```
LOOP
END
```

```
julien@holberton:/proc/7972$ cat maps

...
00901000-00943000 rw-p 00000000 00:00 0                                [heap]
...
julien@holberton:/proc/7972$
```

-> 00901000-00943000 rw-p 00000000 00:00 0 [heap]

The beginning of the heap is still the same, but the size has increased upwards from 00922000 to 00943000.

# The Address Space Layout Randomisation (ASLR)

You may have noticed something “strange” in the /proc/pid/maps listing above, that we want to study:

The program break is the address of the first location beyond the current end of the data region – so the address of the first location beyond the executable in the virtual memory. As a consequence, the heap should start right after the end of the executable in memory. As you can see in all above listing, it is NOT the case. The only thing that is true is that the heap is always the next memory region after the executable, which makes sense since the heap is actually part of the data segment of the executable itself. Also, if we look even closer, the memory gap size between the executable and the heap is never the same:

*Format of the following lines: [PID of the above maps listings]: address of the beginning of the [heap] – address of the end of the executable = memory gap size*

- [3718]: 01195000 – 00602000 = b93000
- [3834]: 024d6000 – 00602000 = 1ed4000
- [4014]: 00e70000 – 00602000 = 86e000
- [4172]: 01314000 – 00602000 = d12000
- [7972]: 00901000 – 00602000 = 2ff000

It seems that this gap size is random, and indeed, it is. If we look at the ELF binary loader source code (fs/binfmt\_elf.c) we can find this:

```

        if ((current->flags & PF_RANDOMIZE) && (randomize_va_space > 1)) {
            current->mm->brk = current->mm->start_brk =
                arch_randomize_brk(current->mm);
#ifdef compat_brk_randomized
            current->brk_randomized = 1;
#endif
        }

```

where `current->mm->brk` is the address of the program break. The `arch_randomize_brk` function can be found in the `arch/x86/kernel/process.c` file:

```

unsigned long arch_randomize_brk(struct mm_struct *mm)
{
    unsigned long range_end = mm->brk + 0x02000000;
    return randomize_range(mm->brk, range_end, 0) ? : mm->brk;
}

```

The `randomize_range` returns a start address such that:

```

[..... <range> .....]
start                end

```

Source code of the `randomize_range` function (`drivers/char/random.c`):

```

/*
 * randomize_range() returns a start address such that
 *
 * [..... <range> .....]
 * start                end
 *
 * a <range> with size "len" starting at the return value is inside in the
 * area defined by [start, end], but is otherwise randomized.
 */
unsigned long
randomize_range(unsigned long start, unsigned long end, unsigned long len)
{
    unsigned long range = end - len - start;

    if (end <= start + len)
        return 0;
    return PAGE_ALIGN(get_random_int() % range + start);
}

```

As a result, the offset between the data section of the executable and the program break initial position when the process runs can have a size of anywhere between 0 and 0x02000000. This randomization is known as Address Space Layout Randomisation (ASLR). ASLR is a computer security technique involved in preventing exploitation of memory corruption vulnerabilities. In order to prevent an attacker from jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the positions of the heap and the stack.

## The updated VM diagram

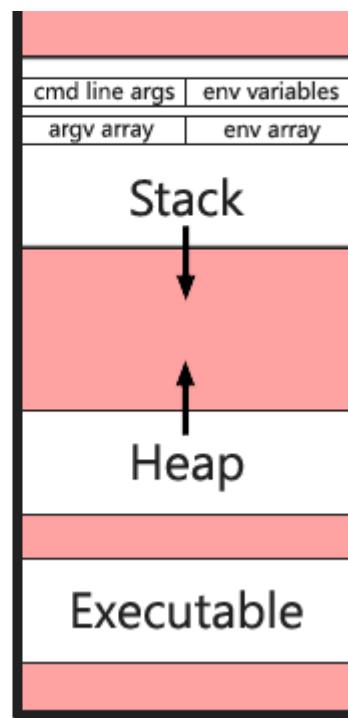
With all the above in mind, we can now update our VM diagram:



High address

Low address

random brk offset {



`malloc(0)`

Did you ever wonder what was happening when we call `malloc` with a size of `0`? Let's check! (10-main.c)

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

/**
 * pmem - print mem
 * @p: memory address to start printing from
 * @bytes: number of bytes to print
 *
 * Return: nothing
 */
void pmem(void *p, unsigned int bytes)
{
    unsigned char *ptr;
    unsigned int i;

    ptr = (unsigned char *)p;
    for (i = 0; i < bytes; i++)
    {
        if (i != 0)
        {
            printf(" ");
        }
        printf("%02x", *(ptr + i));
    }
    printf("\n");
}

/**
 * main - moving the program break
 *
 * Return: EXIT_FAILURE if something failed. Otherwise EXIT_SUCCESS
 */
int main(void)
{
    void *p;
    size_t size_of_the_chunk;
    char prev_used;

    p = malloc(0);
    printf("%p\n", p);
    pmem((char *)p - 0x10, 0x10);
    size_of_the_chunk = *((size_t *)((char *)p - 8));
    prev_used = size_of_the_chunk & 1;
    size_of_the_chunk -= prev_used;
    printf("chunk size = %li bytes\n", size_of_the_chunk);
    return (EXIT_SUCCESS);
}

```

```

julien@holberton:~/holberton/w/hackthevm3$ gcc -Wall -Wextra -pedantic -Werror 10-
main.c -o 10
julien@holberton:~/holberton/w/hackthevm3$ ./10
0xd08010
00 00 00 00 00 00 00 00 21 00 00 00 00 00 00 00
chunk size = 32 bytes
julien@holberton:~/holberton/w/hackthevm3$

```

-> `malloc(0)` is actually using 32 bytes, including the first 0x10 bytes.

Again, note that this will not always be the case. From the man page (`man malloc`):

NULL may also be returned by a successful call to `malloc()` with a size of zero

## Outro

We have learned a couple of things about `malloc` and the heap. But there is actually more than `brk` and `sbrk`. You can try `malloc`'ing a big chunk of memory, `strace` it, and look at `/proc` to learn more before we cover it in a next chapter 😊

Also, studying how `free` works in coordination with `malloc` is something we haven't covered yet. If you want to look at it, you will find part of the answer to why the minimum chunk size is 32 (when we ask `malloc` for 0 bytes) vs 16 (0x10 in hexadecimal) or 0.

As usual, to be continued! Let me know if you have something you would like me to cover in the next chapter.

## Questions? Feedback?

If you have questions or feedback don't hesitate to ping us on Twitter at [@holbertonschool](#) or [@julienbarbier42](#).

*Haters, please send your comments to `/dev/null`.*

Happy Hacking!

## Thank you for reading!

As always, no-one is perfect (except [Chuck](#) of course), so don't hesitate to [contribute](#) or send me your comments if you find anything I missed.

## Files

[This repo](#) contains the source code (`naive_malloc.c`, `version.c` & `"X-main.c"` files) for programs created in this tutorial.

## Read more about the virtual memory

Follow [@holbertonschool](#) or [@julienbarbier42](#) on Twitter to get the next chapters!

- Chapter 0: [Hack The Virtual Memory: C strings & /proc](#)
- Chapter 1: [Hack The Virtual Memory: Python bytes](#)
- Chapter 2: [Hack The Virtual Memory: Drawing the VM diagram](#)
- Chapter 3: [Hack the Virtual Memory: malloc, the heap & the program break](#)